TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

MODULARIZACE SIMULÁTORU PORUCH ČÍSLICOVÝCH OBVODŮ

Bakalářská práce

Studijní program: B2646 – Informační technologie
Studijní obor: 1802R007 – Informační technologie

Autor práce: **Tomáš Červený**
Vedoucí práce: Ing. Jiří Jeníček, Ph.D.

Liberec 2014

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

FAULT SIMULATOR MODULARIZATION

Bachelor thesis

Study programme: B2646 – Information Technology
Study branch: 1802R007 – Information Technology

Author: **Tomáš Červený**
Supervisor: Ing. Jiří Jeníček, Ph.D.

Liberec 2014

Tento list nahraďte
originálem zadání.

Prohlášení

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu TUL.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum:

Podpis:

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií
Akademický rok: 2013/2014

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Tomáš Červený**
Osobní číslo:	**M10000135**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Modularizace simulátoru poruch číslicových obvodů**
Zadávající katedra:	**Ústav informačních technologií a elektroniky**

Zásady pro vypracování:

1. Seznamte se s problematikou testování poruch číslicových obvodů.
2. Analyzujte existující simulátor poruch číslicových obvodů a navrhněte vhodnou objektovou strukturu s vhodným využitím standardních knihoven C++.
3. Simulátor přepište z C do C++.
4. Nový simulátor otestujte na správnost výsledků a na rychlost.

Rozsah grafických prací:	**Dle potřeby dokumentace**
Rozsah pracovní zprávy:	**cca 30 stran**
Forma zpracování bakalářské práce:	**tištěná/elektronická**

Seznam odborné literatury:

[1] **Hlavička, J.: Diagnostika a spolehlivost. Praha, Vydavatelství ČVUT, 1998, ISBN 8001018466**

[2] **Prata, S.: Mistrovství v C++, Computer press, 2007, EAN 9788025117491**

Vedoucí bakalářské práce:	**Ing. Jiří Jeníček, Ph.D.** Ústav informačních technologií a elektroniky
Datum zadání bakalářské práce:	**12. září 2013**
Termín odevzdání bakalářské práce:	**16. května 2014**

V. Kopecký

prof. Ing. Václav Kopecký, CSc.
děkan

Technická univerzita v Liberci - 14 -
L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
vedoucí ústavu

V Liberci dne 12. září 2013

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu TUL.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Bakalářskou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Současně čestně slibuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum 12. 9. 2014

Podpis:.....

Abstrakt

Účelem této práce je v teoretické části se seznámit s problematikou testování poruch číslicových obvodů a získané vědomosti pak aplikovat v praktické části, při analýze simulátoru chyb číslicových obvodů Fsim, který je psán v programovacím jazyce C. Tento simulátor pak bude nutné převést do objektového programovacího jazyka C++. K tomuto převodu jsou dva hlavní důvody, první z nich je zlepšení přehlednosti kódu a usnadnění další práce na tomto programu. To se má docílit nahrazením velkého množství různých struktur, v tomto programu použitých, prostředky ze standartních knihoven C++, které v prostém jazyce C chybí. Jiné struktury pak budou přepracovány na třídy a zapouzdřeny, z důvodů lepšího řízení přístupu k jejich vnitřním proměnným. Dále by se také měla zlepšit přehlednost tohoto programu převedením mnoha použitých maker na funkce, což by mělo zlepšit typovou bezpečnost a snížit riziko nechtěně destruktivního zásahu do programu.

Druhým důvodem pro transformaci do C++ je příprava programu na další práci na něm, hlavně na jeho převedení na vícevláknový program, který byl schopen zpracovávat ve více různých vláknech více různých obvodů najednou. Tomuto procesu brání hlavně široké použití globálních a statických proměnných pro většinu hodnot, týkajících se dané simulace a jejího stavu. Pokud by se tak spustilo více simulací najednou, navzájem by si přepisovaly svoje hodnoty, a tím pádem by nemohly fungovat. Tento problém bude řešen zapouzdřením proměnných řídících danou simulaci do jedné třídy, která pak bude spravovat celou simulaci.

Výsledný program pak bude nutné otestovat na správnost výsledků vůči původnímu programu na poskytnutých testovacích souborech obvodů, testovacích vektorů a seznamů chyb. Při tomto testu musí nový program dávat stejné výsledky jako program původní.

Klíčová slova: simulace poruch, C, C++, číslicové obvody

Abstract

Purpose of this task in the theoretical part is to become acquainted with problematic of fault simulation on digital circuits. Then use this knowledge in the practical part of this task to analyze fault simulator of digital circuits called Fsim, which is written in programing language C. Then it is necessary to translate this fault simulator to object programing language C++. There are two main reasons for this. The first one is to make simulator's code more transparent and make it easier to work with for future development. This should be accomplished through usage of features of standard C++ library that are missing from plain C. Other structures will be transformed into C++ classes and encapsulated, for the purpose of easier management of access to inner variables. Transparency of this code could be enhanced by transforming this program's many macros into functions, which also enhances its type safety and decreases chance of unwantedly destructive interference with the code.

Second main reason for transforming this program into C++ is preparation of this program for transformation into multi-thread program, which will be able to use multiple threads to simulate multiple circuits at once. This process is mainly prevented by wide use of global and static variables for majority of values responsible for managing running simulation and holding its status. If multiple simulations were run at the same time they would overwrite each other's variables and would not be capable of functioning properly. This problem will be solved by encapsulating variables responsible for managing simulation into one class that will be responsible for running whole simulation.

The final program then must be tested to determine whether it gives the same results as the original program on provided testing files of circuits, testing vectors and fault lists.

Key words: fault simulation, C, C++, digital circuits

Obsah

1 Úvod

Účelem této práce je seznámení se s problematikou simulace poruch číslicových obvodů, analyzovat existující simulátor poruch číslicových obvodů FSIM a navrhnout pro něj objektovou strukturu. Praktickou částí této práce je přepsání tohoto simulátoru z jazyka C do jazyka C++. Nakonec je nutné otestovat takto vytvořený nový simulátor proti původnímu na správnost výsledků a porovnat jejich rychlosti.

Hlavním účelem této práce je připravit půdu pro další práci na tomto programu jednak úpravou do objektového programování a hlavně ho připravit na převedení na vícevláknový, aby bylo možné zpracovávat testované soubory paralelně na více jádrových počítačích. Hlavní překážkou vícevláknovému běhu tohoto simulátoru je široké nasazení globálních proměnných, které, v případě že by běželo několik simulací najednou, by byly navzájem přepisovány různými simulacemi. Tento program je také v některých částech velice nepřehledný, hlavně kvůli využití vlastních nadefinovaných struktur pro uložení polí, z nichž každá tento problém řeší jinak.

V přepracované verzi by mělo dojít ke zjednodušení především použitím standardních knihoven jazyka C++, které nabízejí stejnou a častěji širší funkčnost a jsou bezpečnější a průhlednější při užití. Další struktury specifické pro tento program budou převedeny na objekty a zapouzdřeny pro lepší řízení přístupu k nim. V programu je také použito velké množství různých maker, která jsou často i přes deset řádků dlouhá, to vytváří problém s typovou bezpečností, protože makra nijak nekontrolují typy vstupních proměnných, a také to zhoršuje přehlednost kódu.

Problém statických a globálních proměnných bude řešen vytvořením nové třídy, která zapouzdří celý běh simulace a zaručí tak, že pokud ve vícevláknovém programu poběží několik simulací najednou, nedojde ke kolizi různých simulací a vzájemnému přepisování statických dat.

1.1 C++

C++ je programovací jazyk podporující procedurální, objektově orientované a generické programování. Používá se v široké škále odvětví od složitějších embedded systémů a jader operačních systémů, přes desktopové a serverové aplikace, vesmírné družice, až po počítačové hry. Stejně jako u jazyka C je počátek tohoto jazyka v Bellových laboratořích kde se Bjarne Stroustrup od roku 1979 pokoušel vytvořit jazyk

podobně efektivní a flexibilní jako C, který by ale podporoval i funkce vyšší úrovně programování jako například podporu objektů. Od té doby se jazyk C++ vyvíjí, jako standard C++98 byl přijat v roce 1998 a nejaktuálnější verze standardu je C++11.

V této verzi standardu byly přidány funkce jako automatická detekce typu, kdy je proměnné přiřazen typ až s přirazením hodnoty a proměnná je pak stejného typu jako hodnota do ní přiřazovaná. Tato funkce je využívána hlavně v kontejnerech, kde se až při přeložení kódu určí, jaký typ proměnné obsahuje. S tím je spojená funkce pro určení, jakého typu je daná proměnná. V této verzi je také často problematické makro NULL nahrazeno novým nullptr což je hodnota neinicializovaného pointeru. Další přidanou funkcí jsou „chytré“ pointery uniquepointer a sharedpointer. Uniquepointer je používán jako jediný pointer na daná data. Pokud je tento pointer smazán, jsou automaticky smazána i data. Shardpointer je jeden z několika pointerů na daná data a ta se automaticky smažou po smazání posledního ze sharedpointerů. Asi nejdůležitějším přídavkem C++11 jsou knihovny pro práci s vlákny, které programu dovolují využívat vícejádrovost moderních procesourů.

Syntaxe jazyka C++ je velice podobná syntaxi jazyka C. Oba jazyky jsou do určité míry kompatibilní, ale liší se v několika klíčových detailech, jako například typová kontrola. Tyto jazyky mají sice statické typy proměnných, ale C dovoluje implicitní konverze mezi některými typy, jako například char a int. C++ je v tomto ohledu mnohem striktnější a konverze se musí vždy jasně specifikovat. Tento a jiné rozdíly v klíčových funkcích zabrání složitějším programům napsaným v C být zkompilovány v C++ bez drobných úprav. Kompatibilita jazyků rozděluje programátory na dva tábory. Jeden tvrdí, že by se měla kompatibilita držet na co největší úrovni, kvůli snadnému přenosu programů, druhý tvrdí, že pokusy o větší kompatibilitu by neměli brzdit samostatný vývoj jednotlivých jazyků.

1.2 Standartní knihovny jazyka C++

Důležitou součástí standardů C++ jsou i standardní knihovny C++, což je soubor tříd a funkcí pro běžné užití při programování. Základem jsou přepracované standardní knihovny z jazyka C, které pomáhají s nejzákladnějšími potřebami při programování, jako například matematické funkce, jednoduchá práce s časem, limity pro jednotlivé číselné datové typy nebo základní vstupní a výstupní operace se soubory. V C++ jsou

navíc přidány knihovny pro vstupní a výstupní datové proudy, které ulehčují práci se soubory nebo s řetězci. Další důležitou skupinou knihoven jsou knihovny určené pro obsluhu vláken, ty dovolují synchronizaci vláken, uzamykání proměnných, a řízení přístupu ke sdíleným proměnným.

1.3 Kontejnery

Další skupinou standardních knihoven jsou kontejnery, což jsou šablony pro objekty určené k práci s dynamickými poli, kde každý z kontejnerů řeší problém s dynamickým polem různým způsobem.

Původní program Fsim je psán v čistém jazyce C a s použitím pouze standardních knihoven tohoto jazyka. Jelikož v čistém C nejsou žádné datové struktury sloužící jako kontejner, jsou v programu doimplementovány struktury, které tuto funkčnost nahrazují. Těchto struktur je tam několik od jednoduchého pole s uloženou velikostí až po složitou implementaci mapy.

1.3.1 Množina

Množina (anglicky set) je dynamické asociativní tříděné pole hodnot, kde se každá z hodnot může vyskytovat pouze jednou. Jelikož je tento kontejner tříděný, všechny hodnoty do něj vkládané musí vždy být porovnatelné. Asociativní pole jsou v jazyce C++ optimalizovány na rychlé vyhledávání v poli, jehož hodnoty nemusí být číselné, a na přístupu k datům pomocí klíče, ne jejich pozice v poli. Rozšířením množiny je multiset, ve kterém je oproti množině možné mít více záznamů se stejnou hodnotou. Problémem tohoto rozšíření ale je, že při použití jiného třídícího algoritmu mohou být jednotlivé záznamy v rámci stejné hodnoty tříděny různě.

Samotný kontejner množina v programu Fsim použit není, jelikož se nepodobá žádné z proměnných použitých v původním programu, ale je v něm použitá jeho rozšířená verze, mapa.

1.3.2 Mapa

Další z asociativních tříděných dynamických polí je mapa, také známá jako slovník. Mapa funguje podobně jako množina, ale obsahuje páry objektů typu klíč-hodnota a je tříděná podle hodnot klíče, kterému je pak přiřazena odpovídající hodnota.

Klíč musí být v mapě unikátní, hodnota se však může vyskytovat i vícekrát. Podobně jako u množiny má mapa rozšíření multimap které dovoluje mít v mapě jeden klíč vícekrát.

V programu Fsim je mapa využívána při načítání jednotlivých hradel z netlistu (popisu obvodu), jako seznam hradel, kde je klíčem symbol hradla (což je název daný souborem netlistu) a hodnotou je ukazatel na samotné hradlo. To zajišťuje, že program načte dané hradlo pouze jednou a v obvodu nebudou vznikat duplicity. V původním programu je mapa implementována pomocí struktury a několika funkcí, které ji obsluhují. Ve verzi převedené do C++ je použit kontejner mapa ze standardní knihovny, který je lépe optimalizovaný a díky zapouzdření není náchylný na chyby v použití.

1.3.3 Fronta

Fronta (v C++ queue) je spojový seznam určený pro použití ve FIFO datové struktuře, kde jsou data přidávána na jeden konec spojového seznamu a odebírána z druhého. Tento kontejner není určen pro operace na datech uprostřed fronty. Fronta je většinou implementována jako obousměrný spojový seznam, ale může být implementována i jako jednosměrný spojový seznam s ukazatelem na oba konce fronty.

1.3.4 Zásobník

Zásobník (v C++ stack) je kontejner podobný frontě, s tím rozdílem že používá strukturu dat LIFO a přidávání a odebírání hodnot se tak odehrává na stejném konci seznamu. Zásobník je většinou implementován jako pole nebo jako jednosměrný seznam.

V programu Fsim je mnoho míst, kde se dynamické pole používá jako zásobník nebo fronta. Ve většině případů je ale nutné na jiném místě programu buď přistupovat k datům uprostřed pole, nebo přidávat a odebírat hodnoty z obou konců. Proto není ani fronta ani zásobník použit v přepracované verzi programu a místo nich jsou použité kontejnery vektor a obousměrný spojový seznam.

1.3.5 Vektor

Vektor je dynamické pole s rychlým přístupem ke všem hodnotám v něm uloženým. Je implementován jako pole, které se dynamicky rozšiřuje a smršťuje podle toho, jak jsou do něj přidávány a odebírány hodnoty, z čehož vyplívají jeho silné i slabé stránky. Mezi jeho výhody patří rychlý přístup ke kterékoli hodnotě v poli pomocí její

pozice a rychlý je i při postupném procházení všech hodnot v něm uložených a při přidávání a odebírání hodnot z konce vektoru. Nevýhodou je, že při přidání nebo odebrání hodnoty z jiného místa než z konce vektoru se musí celý vektor překopírovat s danou změnou. Tato operace je dost výpočetně náročná a tudíž pomalá.

V programu Fsim přepracovaném do C++ je vektorem nahrazena většina polí v kódu. Tím se zamezí problémům s pevně danými maximálními velikostmi polí a nutností počítat velikost před alokováním paměti. Vektory jsou tak použity pro seznam hradel, seznam chyb a seznamy vstupů a výstupů jednotlivých hradel a další seznamy. Vektory jsou pro seznamy použity proto, že do těchto seznamů se hodnoty téměř vždy přidávají nebo odebírají na konci a často se procházejí a některá funkce je spouštěna pro každý ze záznamů. V původní verzi programu jsou také implementovány struktury, které kromě pole hradel nebo poruch obsahují také velikost pole, a jsou k nim i makra přidávající funkce push a pop pro manipulaci s nimi. Alokace pole v této struktuře je statická. Tyto struktury jsou v přepracované verzi nahrazeny vektory, které stejně jako struktury z původního programu mají jak možnost procházet data uvnitř tak i rychle přidávat a odebírat hodnoty z konce.

1.3.6 Obousměrný spojový seznam

Obousměrný spojový seznam (v C++ list) je spojový seznam implementovaný tak, že každý záznam má u sebe ukazatel na předchozí a následující záznam. Díky této implementaci je přidávání a odebírání prvků mimo konce podstatně rychlejší než u vektoru. Vektor má zase rychlejší přístup k položkám uprostřed seznamu, protože je jeho obsah uložen na jedné souvislé části paměti. Z toho vyplývá horší rychlost spojového seznamu při postupném procházení všech záznamů v něm uložených.

V programu Fsim je obousměrný spojový seznam použit pro jednu proměnnou nazvanou Stack, která, jak napovídá její název, je nejčastěji používána jako zásobník, ale v jiných místech je užitá jako fronta, a proto nebylo možné pro ni použít ani jednu ze specializovanějších verzí a bylo nutné použít obecný spojový seznam. Ten má ale stále dobrý výkon při odebírání i přidávání záznamů na oba konce seznamu. Širšímu použití spojového seznamu v programu brání hlavně jeho složitější a výpočetně náročnější přístup k hodnotám uprostřed seznamu.

1.4 ATPG

Automatický generátor testovacích vektorů (anglicky **A**utomatic **T**est **P**attern **G**enerator) je program určený k vytváření testovacích vektorů, které automatickému testovacímu nástroji dovolí zjistit rozdíl mezi funkčním a vadným výrobkem ihned po dokončení, nebo pomoci s určením příčiny zjištěné chyby. Ideálně by byl obvod testován všemi možnými kombinacemi vstupů. Tomuto typu testování se říká triviální test, ale kvůli exponenciální složitosti takovéhoto testování je pro větší obvody prakticky nereálné. Z tohoto důvodu se používají programy ATPG k omezení počtu testovacích vektorů a tedy ke snížení výpočetní náročnosti. Kvalitu vygenerovaných testovacích vektorů pak určí procento odhalených chyb v daném obvodu. Výstupní hodnoty obvodu při bezchybném průchodu se nazývají očekávané výstupní hodnoty. Chyba je detekována v případě, že se hodnota obvodu s vloženou chybou liší od očekávaných výstupních hodnot daného obvodu. ATPG nemusí najít danou chybu v jednom ze dvou případů. První z nich je, když neexistuje žádný vektor, který by takovou chybu detekoval. Toto může nastat například u obvodů se zálohou, které jsou od začátku koncipovány tak, aby žádná chyba nemohla sama změnit kterýkoli z výstupů. Druhý případ je ten, že takový vektor sice existuje, ale generátor jej nenalezne, protože vzdá hledání z časových důvodů. Problém ATPG je NP-úplným problémem a hledání řešení tak může být pro větší obvody velice výpočetně náročné.

1.5 Defekt, chyba, porucha

Defekt je fyzický problém na obvodu, jako například zkrat na vodiči, přerušený vodič nebo poškozená součástka. Chyba je fyzickým projevem defektu v obvodu, takový projev může například být to, že se do hradla nedostává signál nebo že se tam dostává chybný signál. Nejčastější defekty v obvodech jsou, kvůli způsobu vnitřního řešení funkčnosti CMOS, zkraty mezi jednotlivými vodiči a buďto uzemněním nebo napájením. Mezi další časté chyby patří přerušení některého z vodičů nebo zkrat mezi dvěma datovými vodiči.

Porucha se pak používá pro namodelování chyby při simulaci. Nejčastějším modelem pro poruchy je pak model stuck-at, kdy se předpokládá, že všechny chyby v obvodu boudou způsobeny tím, že je daný vodič „zaseknut“ na jedné z logických hodnot. Toto je nejčastěji způsobeno zkratem se zemnícím nebo naopak napájecím

vodičem. U tohoto modelu, který je dlouho nejpoužívanějším modelem chyb, se předpokládá, že jakákoli jiná chyba se někde projeví jako porucha stack-at. Tento model se používá pro kombinační obvody a části sekvenčních obvodů, které se dají odpojit od pamětí. Tento model také předpokládá, že dojde vždy jen k jedné poruše najednou a pro obvod s n spoji existuje 2n možností různých poruch.

Program Fsim, který je předmětem této práce, také používá model stuck-at pro jeho jednoduchost a možnost namodelování většiny možných chyb pomocí stuck-at poruch.

Dalšími druhy chyb jsou například chyby transistorů, kdy je některý z transistorů stále otevřen nebo naopak zavřen, nebo chyby přemostění, způsobené zkratem mezi jednotlivými vodiči, tyto se simulují vložení hradla AND nebo OR do obvodu v daném místě. Tyto chyby se do poruch modelují pomocí jiných, méně častých, modelů poruch.

Chyby se také dělí na stálé a dynamické, dynamické chyby se projevují pouze za určitých podmínek a jsou obtížněji detekovatelné a modelovatelné, zatímco stálé chyby jsou přítomné vždy a lze je tak jednodušeji objevit.

1.5.1 Vzorkování poruch

Protože možností pro poruchu ve větších obvodech je velmi velké množství a testování pro všechny z nich je velice výpočetně náročné, často se používá náhodný výběr určitého množství z nich, čemuž se říká vzorkování poruch. Poté se z pokrytí vzorkovaných poruch daným testem odhaduje pokrytí všech možných poruch. Výhodou tohoto postupu je pak podstatně rychlejší vyhodnocení, které navíc nezávisí na počtu všech možných poruch, ale na velikosti vzorku. Získáme však minimální množství informací o nedetekovaných poruchách a přesnost výsledku závisí na náhodném výběru.

Program Fsim vzorkování sám o sobě nepodporuje, je ale možné načítat seznam testovaných poruch ze souboru, což se pak používá pro zrychlení simulace, pokud je nutné testovat pouze určité poruchy.

1.5.2 Událostmi řízená simulace

Událostmi řízená simulace je efektivní způsob simulování číslicových obvodů založený na zachycení jakékoli změny hodnoty v obvodu, takzvané události. Jakákoli událost může pak spustit několik dalších událostí, přičemž první událost je vždy změna na primárních vstupech. Každá událost na vodiči přidá na seznam aktivity všechna

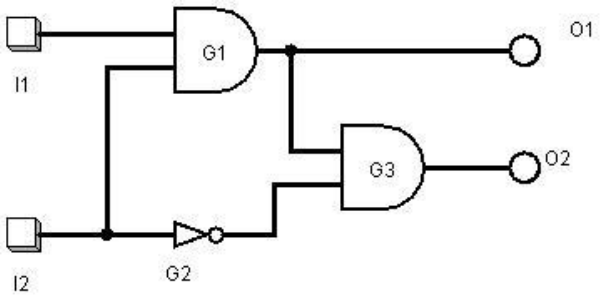
hradla připojená na tento vodič, hradla ze seznamu aktivity se pak postupně odebírají a vyhodnocují. Pokud se hodnota výstupu hradla změní, jsou na seznam aktivity přidány všechny hradla, které jsou na něj připojeny. Proces vyhodnocování skončí, jakmile je seznam aktivity prázdný. Na rozdíl od metod, ve kterých se obvod popíše v kódu programovacího jazyka a zkompiluje se, se v událostmi řízené simulaci při jakékoli změně nemusí vždy simulovat celý obvod znovu, ale simuluje se vždy jen ovlivněná část.

Program Fsim využívá specifickou verzi tohoto algoritmu, kde proběhne pouze jeden průchod obvodu podle podobných pravidel jako událostmi řízený průchod, při to se projdou všechny hradla od primárních vstupů až po výstupy a podle pořadí v jakém se tato hradla vyhodnocovala, se seřadí do seznamu. Seznam se poté prochází postupně, a není nutné, při každém běhu vyhodnocovat pořadí v jakém jsou hradla vyhodnocena.

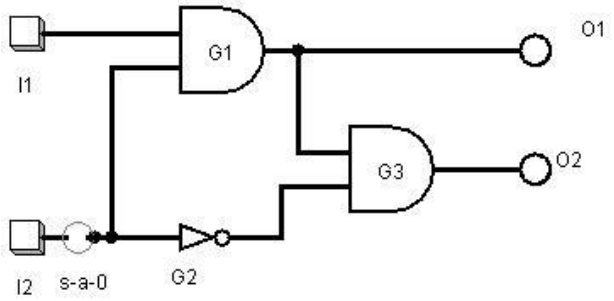
1.6 Algoritmy pro testování poruch číslicových obvodů

1.6.1 Sériový

Sériový algoritmus pro testování poruch je založen na událostmi řízeném simulování, při kterém se nejdříve nasimuluje bezporuchový obvod, a hodnoty bezporuchových výstupů se zaznamenají. Pro každou poruchu pak nejdříve danou poruchu vloží do netlistu, pak postupně zkouší všechny poskytnuté testovací vektory a postupně prochází události na obvodu, a jakmile dojde na kterémkoli primárním výstupu k události lišící se od bezporuchového obvodu, porucha je odhalena a algoritmus zastaví testování vektorů a pokračuje na další poruchu. Tento algoritmus je ze všech nejjednodušší na implementaci a je nenáročný na paměť. Další výhodou je pak jeho schopnost simulovat většinu možných poruch včetně těch analogových. Nevýhodou tohoto algoritmu je velké množství redundantních výpočtů a s tím spojená výpočetní náročnost, která ztěžuje simulaci větších obvodů.



Obrázek 1 Bezporuchový obvod



Obrázek 2 Poruchový obvod s poruchou stuck-at-0 na vstupu I2

1.6.2 Příklad sériové simulace

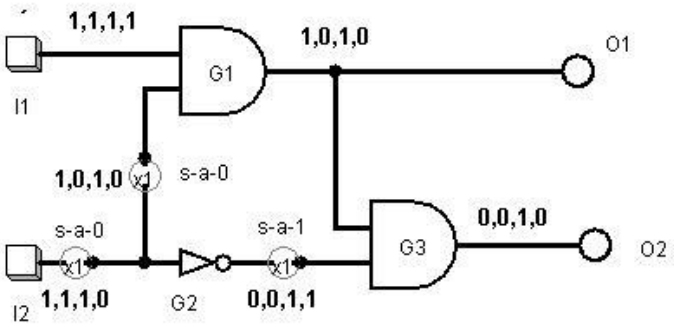
Na prvním obrázku je bezporuchový obvod a na druhém je stejný obvod s poruchou stuck-at-0 na vodiči I2. Tento obvod má dva vstupy: I1 a I2 a dva hlavní výstupy: O1 a O2. Pro testovací vektor (1,1) má bezporuchová simulace výstup (0,1). Událostí je v tomto případě porucha stuck-at-0 (s-a-0) na vodiči I2. Ta při daném vstupním vektoru změní hodnotu vodiče z 1 na 0. Tato událost přidá na seznam aktivity hradla G1 a G2. Nejdříve se vyhodnotí hradlo G1. Jeho bezporuchový výstup je 1, ale změna na jeho vstupu ho změní na 0, tato změna přidá na seznam výstup O1 a hradlo G3. Další v seznamu aktivity je hradlo G2. Jeho bezporuchová hodnota je 0 a ta se v poruchové simulaci změní na 1. Tato změna by měla na seznam aktivity přidat hradlo G3, ale to už tam je. První na seznamu aktivity je teď výstup O1. Jeho vstup se změnil z bezporuchové 1 na 0, tím je porucha odhalena a zjistilo se, že porucha s-a-0 na vodiči I2 je odhalitelná vektorem (1,1). Porucha se vyškrtne ze seznamu testovaných poruch a testování pokračuje s další poruchou. V tabulce je znázorněno, jak by simulace pokračovala, kdyby nebyla přerušena po detekování poruchy.

Čas	událost	Seznam aktivity
0	I2 (1 -> 0)	G1, G2
1	G1 výstup (1 -> 0)	G2, O1, G3
2	G2 výstup (0 -> 1)	O1, G3
3	O1 (1 -> 0) porucha detekována	G3
4	G3 (nezměněno)	

1.7 Paralelní

Paralelní algoritmus se aplikuje na obvody popsané v jazyce, který je možné zkompilovat a spustit v počítači, např. prosté C. Vstupy a výstupy jsou převedeny na proměnné a hradla se pak zapisují jako jednotlivé bitové příkazy daného jazyka. Tento kód je pak spouštěn pro každý testovací vektor v několika průchodech, kdy je při každém průchodu vždy simulováno n obvodů, n-1 obvodů s různými poruchami a jeden bezporuchový obvod pro srovnání, kde n je délka slova počítače, na kterém se testuje. Například na 32 bitovém počítači se simuluje 31 poruchových obvodů najednou a jeden bezporuchový. Tento algoritmus využívá přirozeného paralelismu bitových operací v délce slova v počítačích a je tak zpravidla n-1 krát rychlejší než sériový algoritmus. Další výhodou tohoto algoritmu je malá náročnost na paměť. Je potřeba pouze počet bitů odpovídající délce slova daného počítače pro každý spoj. Nevýhodou je nevhodnost použití tohoto algoritmu pro obvody s časovou závislostí a jinou než booleovskou logikou, a také to, že bezporuchový průchod se simuluje při každém průchodu.

1.7.1 Příklad paralelní simulace



Obrázek 3 Obvod pro paralelní testování poruch

Na obrázku je znázorněn stejný graf jako v případě se sériovou simulací, ale jsou v něm vyznačeny tři různé poruchy. Porucha jedna, s-a-0 na vstupu hradla G1 z I2, porucha dvě, s-a-1 na výstupu z hradla G2 a porucha tři, s-a-0 na vstupu I2. Pro jednoduchost simulace v příkladu uvažujeme počítač s délkou slova 3. U všech vodičů jsou vyznačeny jejich logické hodnoty při vstupním vektoru (1,1). První hodnota odpovídá bezporuchové simulaci, druhá hodnota odpovídá hodnotě při poruše jedna, další při poruše dvě a poslední při poruše tři. Na rozdíl od sériové simulace se zde musí vždy provést simulace všech hradel v obvodu. To se provádí tak, že se na vodičích nasimuluje bezporuchová hodnota, která se pak kopíruje, dokud se nenarazí na poruchu z té dané verze obvodu. Pak se změní hodnota vodiče na hodnotu poruchovou a na

hradlech se pak počítá hodnota z korespondujících hodnot ze slova. Na vstupu I2 jsou první dvě poruchové hodnoty shodné s bezporuchovou, ale poslední hodnota se liší, protože už narazila na poruchu s-a-0 z odpovídající poruchové verze obvodu. Na vstupu hradla G1 se pak projeví jak porucha na vstupu I2, tak i porucha na samotném vstupu do hradla G1, a obě změní bezporuchovou hodnotu 1 na 0. Tím se tyto poruchy projeví i na výstupu hradla G1. Hodnoty výstupu hradla G2 ovlivní jak porucha na samotném výstupu, tak i porucha na vstupu I2. Tyto hodnoty dále pokračují do hradla G3. Tam se účinky první a třetí poruchy neprojeví a na výstupu 02 se projeví pouze druhá porucha. Z obrázku je pak vidět, že vektor (1,1) ukáže všechny tři poruchy, protože první a třetí porucha se projeví na výstupu O1 a druhá porucha se projeví na výstupu O2.

1.8 Deduktivní

V tomto algoritmu se vždy obvod prochází pouze jednou pro každý testovací vektor a simuluje se pouze bezporuchový průchod. Poruchové hodnoty jsou pak dedukovány z bezporuchového chodu. Každé hradlo v obvodu má pak svůj seznam poruch, které můžou ovlivnit jeho výstupy. Obvod se postupně prochází podle pravidel událostmi řízené simulace od primárních vstupů směrem k výstupům. Při průchodu každým hradlem se vytvoří seznam poruch, které mohou ovlivnit jeho výstupy, a podle kterých se patřičně doplní všechny seznamy poruch hradel na toto hradlo navázaných. Seznam poruch se takto šíří dále obvodem. Seznamy poruch na primárních výstupech jsou pak použity jako výsledný seznam detekovaných poruch. Nevýhodou tohoto algoritmu je, že se s ním obtížně simuluje zpoždění na hradlech a pravidla pro šíření poruch v obvodu se obtížně přizpůsobují pro nebooleovskou logiku.

1.9 Souběžný

Stejně jako deduktivní algoritmus je souběžný algoritmus závislý na událostech, které se v tomto algoritmu dělí na dobré události, což jsou události v bezporuchovém obvodě, a poruchové události, které jsou v poruchových obvodech a liší se od bezporuchového průchodu. Na rozdíl od deduktivního algoritmu, se v tomto algoritmu neprovádí průchod pro každý testovací vektor znovu. Proto je tento algoritmus rychlejší než deduktivní algoritmus. Simuluje se pouze jeden bezporuchový průchod obvodem a ty části poruchových průchodů, které se liší od bezporuchového. Na každém hradle je pak seznam jeho poruchových kopií i s typem poruchy, vstupními a výstupními

hodnotami a vnitřními stavy, pokud hradlo nějaké má. Průchod obvodem je podobný jako u deduktivního algoritmu. Tento algoritmus je ze všech nejrychlejší a dovoluje simulovat jakékoli poruchy. Problémem tohoto algoritmu jsou ale velmi vysoké nároky na paměť, které u větších obvodů nedovolují simulovat všechny poruchy v jediném průchodu obvodem.

1.10 Simulace s paralelními vektory a jednou poruchou

V tomto algoritmu jsou testovací vektory zkoušeny na obvodu s jednou poruchou, tedy jedinou událostí. Tímto se sníží výpočetní náročnost na jednu událost. Pouze s jedinou poruchou, tedy jedinou událostí, se pak může simulovat jen ta část obvodu, která je touto událostí ovlivněna. Tento algoritmus je vhodný jedině pro kombinační obvody.

2 Analýza programu Fsim

Fsim je program pro testování souborů s testovacími vektory, který zkouší kolik poruch je daný soubor testovacích vektorů schopen odhalit na daném obvodu. Jako vstup mu slouží soubor .bench, ve kterém je netlist.

2.1 Nastavení programu

Program Fsim může běžet v příkazové řádce s různými parametry. Pokud parametr nezačíná pomlčkou nebo není za parametrem vyžadujícím hodnotu, je automaticky považován za soubor netlistu. Pokud program nemá žádné parametry, vypíše help. Zde je výčet možných parametrů pro program:

- -l je nejdůležitější parametr. Za ním následuje název souboru, do kterého se zapíše výstup z programu. Pokud bude tento parametr chybět, program nic nevypíše.
- -t je parametr pro streamovaný vstup testovacích vektorů. Za ním následuje název souboru se vstupním streamem. Soubory s testovacím streamem jsou popsány dále.
- -T je parametr pro nestreamovaný vstup testovacích vektorů. Za ním následuje název souboru s výpisem vstupních testovacích vektorů. Pokud

není nastaven ani jeden z parametrů –t a –T, program testuje náhodné vektory.

- -n je parametr netlistu. Za ním následuje název souboru s netlistem. Jak vypadá soubor s netlistem je popsáno v samostatné kapitole.
- -h je parametr nápovědy. Pokud je nastaven, program vypíše nápovědu a skončí.
- -f je parametr seznamu poruch. Za ním následuje název souboru se seznamem poruch. Pokud tento parametr není nastaven, program si seznam poruch vytvoří automaticky.
- -U je parametr nedetekovaných poruch. Pokud je nastaven, program do výstupního souboru vypíše jejich seznam.
- -D je parametr detekovaných poruch. Pokud je nastaven, program do výstupního souboru vypíše jejich seznam.
- -S je parametr sumárního výstupu. Pokud je nastaven, program zapíše sumární výstup do výstupního souboru.
- -c je parametr postupného pokrytí. Pokud je nastaven, program na standardní výstup při každém průchodu testovací smyčkou, program vypíše poměr detekovaných poruch.
- -r je parametr limitu testů. Za ním následuje číslo, které se nastaví jako limit počtu testovaných vektorů pro testování náhodnými testovacími vektory. Standardně je tento limit nastaven na 224 náhodných testovacích vektorů, což stačí pouze pro malé obvody. Pokud je nastaven některý z parametrů pro načítání testovacích vektorů ze souboru, limit testů se nastaví na 2147483646, což je limit dostatečný i pro velmi velké obvody.
- -s je parametr randomseed. Za ním následuje číslo, které se nastaví jako počáteční hodnotu generátoru pseudonáhodných čísel pro vytváření náhodných testovacích vektorů. Pokud není nastaven, použije se momentální čas.

2.2 Soubory .bench

Soubory .bench jsou soubory netlistu pro standardy ISCAS85/89, ve kterých je výčet všech hradel v obvodu, jejich názvů, typů a jejich vstupů ve formátu „[název hradla]=[typ hradla (např.: AND OR)]([výčet názvů vstupních hradel oddělený čárkami])“. Primární vstupy jsou vypsány ve formátu „INPUT([název primárního vstupu])“ a primární výstupy ve formátu „OUTPUT([název primárního výstupu])“. Klíčová slova jako INPUT, OUTPUT, AND nebo OR je možné psát buď velkými písmeny anebo malými písmeny, ne však jejich kombinací. Komentáře v souboru vždy začínají křížkem a končí s koncem řádku. Původní verze programu Fsim také vyžadovala, aby byl první řádek zakomentovaný a byl na něm vypsán název obvodu. Nová veze toto nevyžaduje a název obvodu si bere z názvu souboru .bench. Soubor také začíná nejdříve výčtem primárních vstupů, pak výčtem primárních výstupů a poté následuje výčet všech ostatních hradel. To, aby byly nejdříve vyčteny primární vstupy ve správném pořadí, a až pak zbytek hradel, je nutné pro správný běh programu. Na dalším pořadí hradel nezáleží. Program také, kromě konce řádku za komentářem, ignoruje bílá místa a konce řádků, a tak je možné je libovolně vkládat do souboru bez toho, aby to mělo vliv na běh programu.

```
# Edf source: "b02/b02_C.edf"
INPUT(U_REG_SCAN_IN)
INPUT(LINEA)
INPUT(STATO_REG_2__SCAN_IN)
INPUT(STATO_REG_1__SCAN_IN)
INPUT(STATO_REG_0__SCAN_IN)

OUTPUT(U_REG_SCAN_IN)
OUTPUT(U33)
OUTPUT(U38)
OUTPUT(U32)
OUTPUT(U31)

U31 = AND(U35, U37, STATO_REG_2__SCAN_IN)
U32 = NAND(U48, U47)
U33 = NAND(U42, U41)
U34 = NOT(STATO_REG_2__SCAN_IN)
U35 = NOT(STATO_REG_0__SCAN_IN)
```

Obrázek 4 Příklad části souboru .bench

2.3 Soubory .teststream

Pro streamovaný vstup testovacích vektorů je používán typ souboru .teststream, který má velice jednoduchou strukturu, ve které jsou komentáře označeny hvězdičkou a končí s koncem řádku. Ze zbytku souboru program načte všechny jedničky a nuly a ty použije jako vstupní stream pro testovací vektory. Všechny ostatní znaky kromě jedniček a nul jsou ignorovány.

```
* Circuit: b02_C
* Test stream size: 18 (b)
* Scan chain was seeded with following values (internal default):
* 00000
* Test stream:
110010100011110110
```

Obrázek 5 Příklad souboru .teststream

2.4 Soubory .faults

Přípona .faults je v programu použitá pro soubory se seznamy poruch. Tyto soubory mají opět velice jednoduchou strukturu. První možnost je, že je na řádku pouze jméno hradla, na kterém se chyba vyskytuje, a buď /0 nebo /1, označující chybu stuck-at-0 respektive stuck-at-1. V tomto případě se jedná o poruchu na výstupu daného hradla. Druhá možnost je, že na řádku je jméno jednoho hradla, šipka (->), jméno druhého hradla, a až pak označení typu chyby. Toto označuje vstupní chybu druhého hradla na výstupu z prvního hradla po větvení. V tomto typu souboru program, stejně jako u ostatních, ignoruje bílá místa, ale žádné jiné znaky v něm povoleny nejsou, dokonce není možné do něj vkládat žádné komentáře. Pokud je v tomto souboru porušena jeho struktura, program vypíše chybu a ukončí se.

```
U_REG_SCAN_IN /0
U_REG_SCAN_IN /1
U33 /1
U41 /1
LINEA->U40 /0
STATO_REG_2__SCAN_IN->U40 /0
U42 /1
U39 /1
U49 /1
U50 /1
U36 /1
U33 /0
STATO_REG_0__SCAN_IN->U41 /1
U40 /1
STATO_REG_1__SCAN_IN->U42 /1
U35->U39 /1
LINEA->U49 /1
```

Obrázek 6 Příklad části souboru .faults

2.5 Běh programu

2.5.1 Inicializace

V úvodní fázi program načte zadané parametry a pomocí funkce option_set příslušně nastaví proměnné, které pak ovlivňují průchod programem. V C++ verzi pak program přidá všechny nutné hodnoty do konstruktoru třídy Fsim, který pak zodpovídá za celé testování daného obvodu. V původní C verzi hned po nastavení proměnných následuje načtení souboru netlistu.

2.5.2 Načtení netlistu

O načítání souboru netlistu se stará funkce read_circuit z knihovny read_cct.h, která je součástí programu. V této funkci program otevře soubor s netlistem nebo v nové verzi datový proud ze souboru, a postupně z něj znak po znaku čte. Vždy, když narazí na určitý řídící znak (v tomto programu rovná se, kulaté závorky a čárka), program provede příslušnou úpravu. Přidá nové hradlo nebo upraví vlastnosti stávajícího. Hradlo je vždy vytvořeno při prvním použití v netlistu, ať už jako vstupu jiného hradla nebo definice hradla samotného. O to, aby nedocházelo k duplicitám, se stará v nové verzi kontejner map ze standardních knihoven C++, do kterého jsou ukládána hradla, kde jako klíč slouží jejich název, a hodnota je ukazatel na hradlo samotné. V původní C verzi se o toto starala knihovna hash.h, která byla součástí programu. Pak funkce doplní hradlům výstupní seznamy, provede několik kontrol na konzistentnost načtených dat, a pokud dojde k chybě, vrátí -1, jinak vrací počet načtených hradel.

Program pak příslušně upraví primární výstupy a pokračuje do další funkce z knihovny read_cct.h a to levelize. Tato funkce je zodpovědná za správné seřazení hradel v jejich seznamu, aby se pak mohla procházet postupně, a bylo vždy jisté, že jsou všechny vstupy do hradla vyhodnoceny, dříve než se začne vyhodnocovat hradlo samotné. Pak se zkontroluje, jestli jsou všechna hradla připojena k primárním vstupům a funkce skončí.

Poté se inicializují seznamy hradel pro jednotlivé úrovně hloubky obvodu a všem hradlům v oblastech bez rozvětvení se nastaví jejich kořen rozvětvení signálů.

2.5.3 Načtení poruch

V této části programu se načítá seznam poruch, které se budou následně v obvodu hledat. To se může provést dvěma různými způsoby podle toho, jak jsou nastavené parametry pro běh programu. První z nich je, že program načte seznam ze souboru se seznamem poruch podobně, jako se ze souboru netlistu načítají hradla. Jak se soubor postupně načítá, tak se vytvářejí nové poruchy a přiřazují se jim jednotlivé jejich vlastnosti. Toto nastavení se zpravidla používá, pokud se má otestovat, zda nějaká sada vektorů otestuje několik konkrétních poruch. Snížení počtu testovaných poruch by mělo podstatně snížit čas nutný k jejich otestování.

Druhá možnost je, že program sám vytvoří pro daný obvod seznam poruch, kde budou zastoupeny všechny množiny ekvivalentních poruch, které v obvodu můžou být. To se provede tak, že pro každé hradlo, které má více než jeden vstup, se vytvoří příslušné poruchy na všech jeho vstupech, které vedou z rozvětvení. Pro hradla AND a NAND se použijí poruchy stuck-at-1, pro hradla OR a NOR se použije porucha stuck-at-0 a pro ostatní hradla se použijí obě možné poruchy. Pak se podle stejného pravidla přiřadí poruchy na výstupy kořenů rozvětvení signálů a primární výstupy. Poruchy se pak vzájemně prováží s hradly, na nichž se vyskytují, a zkontroluje se, jestli je celý seznam poruch správně vytvořen.

2.5.4 Načítání testovacích vektorů

Simulace samotná probíhá ve smyčce, která vždy začíná načtením testovacích vektorů. Toto se provede podle nastavených parametrů vygenerováním náhodné sady vektorů pomocí standardní funkce random. V tomto případě smyčka běží, dokud počet otestovaných vektorů nedovrší limitu pro náhodný test. Druhou možností je pak načtení ze souboru se streamovaným nebo nestreamovaným vstupem. V tomto případě smyčka běží, dokud nedojdou vektory pro testování. Načítání vektorů z nestreamovaných souborů se již nepoužívá, protože ukládání jednotlivých vektorů pro všechny primární vstupy zvláště u větších obvodů vyžaduje velké množství místa na disku. Načtení vektoru ze streamovaného souboru probíhá tak, že se nejdříve načte celý testovací stream do paměti, v původní verzi se data uloží do staticky alokovaného pole velikosti celého souboru. V přepracované verzi se data postupně ukládají do vektoru. Poté, co jsou všechna data načtena, se při každém běhu pomocí jednoduché implementace

kruhového bufferu postupně vytvářejí samotné testovací vektory, které se pak nastavují jako hodnoty primárních vstupů.

2.5.5 Simulace

Simulaci samotné předchází ještě inicializace, ve které se nastaví seznamy hradel, které se pak v simulaci používají, jako seznam kořenů rozvětvení signálů a primárních výstupů, a seznam všech hradel, na kterých se ještě vyskytují nedetekované poruchy. Také se pro všechny kořeny rozvětvení signálu nastaví jejich kritické cesty. Simulace pak probíhá tak, že pro každou sadu testovacích vektorů se nejdříve provede bezporuchová simulace, která určí bezporuchové hodnoty všech hradel a poté se začne se simulací poruch.

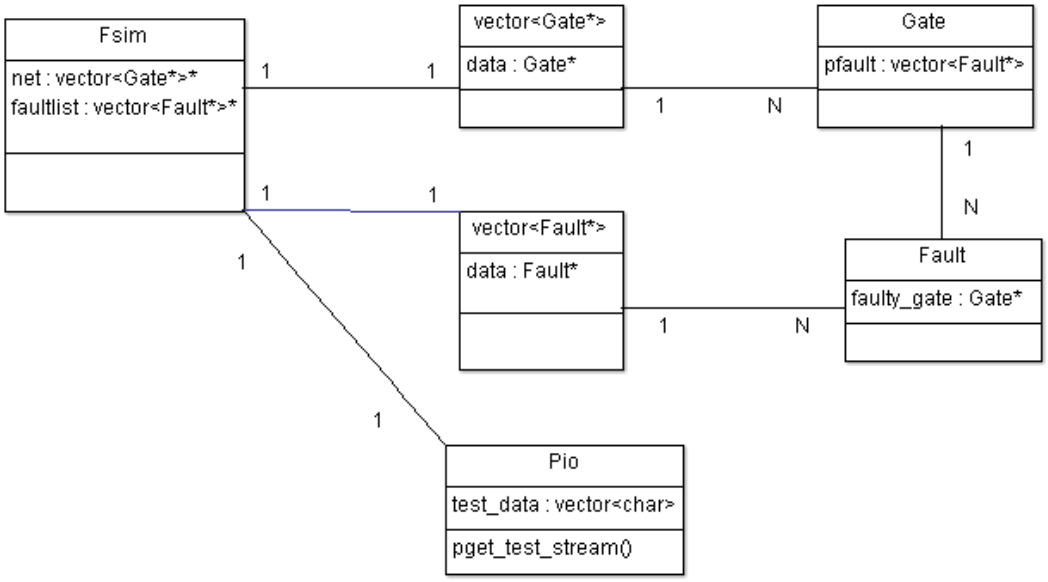
V programu Fsim se simulují všechny poruchy najednou a detekované poruchy se pak označí v seznamu a odebírají se ze seznamů poruch na jednotlivých hradlech. To se provádí tak, že se postupně prochází seznam kořenů rozvětvení signálů a primárních výstupů a pro každou poruchu se pak vyhodnocuje, jaká je její pozorovatelnost na kořeni rozvětvení signálu. Pokud je hradlo primárním výstupem, všechny pozorovatelné poruchy se označí jako detekované a odstraní se ze seznamu poruch na daném hradle.

Po skončení simulace pro daný set testovacích vektorů se zkontroluje, jestli již nebyly detekovány všechny poruchy. Pokud ne, program obnoví seznam hradel s nedetekovanými poruchami a seznam kořenů rozvětvení signálů a primárních výstupů, aby byly připraveny na další běh programu. Smyčka se poté spustí od začátku načtením nových testovacích vektorů.

3 Změny oproti původní verzi

Největší změnou oproti původní verzi je rozdělení hlavní části programu, která byla v původní verzi celá ve funkci main v souboru fsim.c. V nové verzi ve funkci main zůstává pouze část týkající se zpracování všech parametrů. Ta doznala pouze minimálních úprav, hlavně týkajících se změny ukládání názvů souborů z původních polí charů, staticky alokovaných na konstantní délku, na stringy, které využívají pouze tolik paměti, kolik je nutné, a nabízí i dodatečné funkce. Vše ostatní bylo přesunuto do konstruktoru nové třídy Fsim, která také udržuje všechny proměnné týkající se dané simulace, které byly v původní verzi použité jako globální a definované v knihovně atpg. Toto byla knihovna určená právě k definici globálních proměnných používaných

po celém programu a také zde byly definovány různé struktury a velké množství maker. Při práci na tomto programu jsem se snažil používat co nejvíce datové struktury ze standardních knihoven C++ a snížit počet struktur v programu, aby se zlepšila jeho přehlednost a nový programátor nemusel zkoumat, jak se v tomto programu používá ta která struktura.



Obrázek 7 Zjednodušené objektové schéma

Na obrázku je znázorněna velice zjednodušená objektová struktura nové verze programu. Základem této struktury je třída Fsim, ta obsahuje celou řadu různých proměnných, ale z hlediska struktury dat v programu jsou hlavní pouze dvě. Tyto dvě proměnné jsou vektor net, obsahující ukazatele na všechna hradla v načteném obvodu, a vektor fault_list, obsahující ukazatele na všechny poruchy které se mají testovat. Dále má také třída Fsim proměnnou třídy Pio, sloužící k načítání testovacích vektorů. Třída Gate obsahuje, kromě množství jiných proměnných, také vektor poruch na tomto hradle se vyskytujících. Třída Fault naopak obsahuje ukazatel na hradlo, na němž se tato porucha vyskytuje, čímž je s hradlem obousměrně spojena.

3.1 Hradla

Dvě nejdůležitější struktury jsou GATE a FAULT. Proměnné instance struktury GATE drží všechny důležité informace týkající se konkrétního hradla, jako například funkci hradla (např.: AND, NAND nebo OR) vybranou z výčtového typu, pole jeho

vstupních hradel, jeho výstupních hradel, seznam nedetekovaných poruch na něm, nebo pokud je hradlo kořenem rozvětvení signálů, také jeho kritickou cestu. Protože v čistém jazyce C není možné určit délku daného pole, jsou zde uloženy také manuálně nastavované proměnné s délkou jednotlivých polí. Také je zde ukazatel na další hradlo, který slouží jako spojový seznam při načítání hradel ze souboru. Hradla se následně spočítají a uloží do staticky alokovaného pole. Dále jsou zde proměnné, týkající se samotné simulace, jako pozorovatelnost hradla z jeho kořenu rozvětvení signálu nebo jeho výstupní hodnota. Poslední důležitá proměnná zde zmíněná, je ukazatel na strukturu HASH, která bude popsána ve vlastní kapitole.

V nové verzi je struktura GATE změněna na třídu Gate a všechna statická pole jsou nahrazena kontejnery vektor, které lze postupně dynamicky plnit při načítání a není nutné uchovat jejich velikost někde zvlášť. To zlepšuje přehlednost kódu a programátor nemusí stále dávat pozor, jestli v proměnné s velikostí udržuje správnou hodnotu. Všechny vnitřní proměnné jsou zapouzdřeny a jsou přístupné pouze pomocí getterů a setterů. V konstruktoru jsou také inicializovány všechny dynamické seznamy a v destruktoru jsou zase smazány, takže se programátor nemusí starat o alokaci a dealokaci paměti pro tyto seznamy. Ukazatel na další hradlo je již odstraněn, jelikož staticky alokované pole, pro které bylo nutné zjistit počet hradel, bylo nahrazeno kontejnerem vektor, který umožňuje, aby se do něho hradla ukládala rovnou v okamžiku, když jsou načítána.

3.2 Poruchy

Druhou důležitou strukturou v původním programu byla struktura FAULT. Ta uchovává všechny důležité proměnné týkající se dané poruchy: hlavně ukazatel na hradlo, na kterém se daná porucha vyskytuje, typ poruchy, buďto stuck-at-1 nebo stuck-at-0, číselnou hodnotu určující, jestli je daná porucha na výstupu nebo na kterém vstupu je, a také hodnota určující, jestli byla tato porucha již detekována. Struktura také obsahuje ukazatele na předchozí a následující poruchy, které byly použity v seznamech poruch na jednotlivých hradlech.

Struktura FAULT byla také přepracována do nové třídy Fault, která je také zapouzdřena a byly přidány gettery a settery na její proměnné. Ukazatele na další

a předchozí poruchy jsou kompletně odstraněny a jsou nahrazeny kontejnerem vektor na hradle, kde se daná porucha vyskytuje.

3.3 Hash

Třetí důležitou strukturou v původním programu je struktura HASH. Tato struktura obsahuje ukazatel na hradlo, název hradla, klíč, což je číslo vzniklé jako hash názvu hradla a používá se pro rychlejší vyhledávání, a ukazatel na další strukturu HASH. Tímto ze struktur vzniká jednoduchý spojový seznam. V původním programu je tato struktura použitá tak, že v něm je staticky alokované pole těchto spojových seznamů délky 39989. Každý hash je pak vždy uložen do seznamu podle jeho zbytku po dělení právě tímto číslem. K této struktuře také patří její vlastní knihovna hash, která obsahuje čtyři funkce. První z nich slouží jen pro alokaci paměti.

Druhá funkce je určená pro prohledávání seznamu hashů, té se předá jako vstup název hradla. Funkce název zahashuje a pak postupně prochází příslušný spojový seznam a hledá klíč vzniklý zahashováním. Pokud klíč najde, srovná ještě, jestli se opravdu shoduje název hradla s hledaným názvem, pro případ kolize hashů, a pokud se názvy shodují, vrací ukazatel na strukturu HASH s hledaným hradlem. Pokud funkce hledané hradlo nenajde, vrací NULL.

Třetí funkce je určená ke vkládání hradel do seznamu hashů. Tato funkce nejdřív ze znaků v názvu hradla spočítá jeho klíč, vytvoří instanci struktury a tu poté uloží do příslušného seznamu podle jeho zbytkové třídy.

Čtvrtá funkce pro nalezení či vytvoření hashe se pak spouští v hlavním programu. Ta nejdříve spustí druhou funkci, aby zjistila, jestli už hash s hledaným názvem existuje, pokud ano, hash vrátí, v případě že ne, spustí třetí funkci, která vytvoří nový hash a ten pak vrátí.

Celá tato struktura i s její knihovnou je v nové verzi odstraněna a nahrazena kontejnerem map ze standardní knihovny C++. Tento kontejner zajišťuje celou tuto funkčnost pouze přes operátor[] a jednoduchou konstrukci k určení, jestli je daný klíč již v mapě. Mapa je v přepracovaném programu použita tak, že jejím klíčem je string, obsahující název daného hradla a hodnotou je pak ukazatel na samotné hradlo. Programátor tedy nemusí řešit vnitřní funkčnost této struktury a navíc je zajištěno, že

nevznikne problém nechtěným zásahem do datové struktury, jelikož ta je celá zapouzdřena.

3.4 Ostatní struktury

Struktura link je jednoduchý jednosměrný spojový seznam. Ten je v původním programu využíván k ukládání kritické cesty na hradle, pokud toto hradlo je kořen větvení signálu. Tato struktura byla nahrazena v nové verzi kontejnerem vektor, který, stejně jako tato struktura, dovoluje jak rychlé přidávání a odebírání z konce, tak i přístup k proměnným uprostřed.

Struktura FLIST byla jednoduchá struktura, obsahující staticky alokované pole poruch a jeho velikost. Bohužel velikost nebyla s polem nijak provázána, a tak bylo nutné ji udržovat ručně. Tato struktura byla použita pouze jako seznam všech testovaných poruch v obvodu. V nové verzi je tato struktura také nahrazena kontejnerem vektor.

Poslední ze struktur použitých v původní verzi programu je EVENT, což je struktura podobná FLIST, jen obsahuje pole polí hradel místo pole poruch. Tato struktura je použitá jako pole, ve kterém jsou pro každou úroveň hloubky obvodu uložena všechna hradla v dané úrovni. V nové verzi je tato struktura nahrazena kontejnerem vektor, který obsahuje další vektory hradel pro jednotlivé úrovně hloubky. Kontejner vektor zajišťuje veškerou nutnou funkčnost a zároveň, protože se jedná o součást standardních knihoven C++, je programátorovi na první pohled jasné, jak jsou data v této datové struktuře uložená, a jak je možné tuto strukturu používat.

3.5 Změny v načítání hradel

Načítání hradel ze souboru v tomto programu zajišťuje knihovna read_cct. V její funkci read_circuit byla původní pole znaků staticky alokovaná na délku 250 znaků nahrazena stringy, které se alokují dynamicky, tudíž nevzniká problém s dosažením maximální hodnoty a také problém se zbytečným zabíráním nepoužívané paměti. Proměnná pro vytváření seznamu vstupů byla také změněna ze stejných důvodů ze statického pole s délkou 1030 ukazatelů na hradla, na kontejner vektor. Staticky alokované pole pro vytváření seznamu primárních výstupů s délkou 11000 ukazatelů na hradla bylo úplně odstraněno a program už rovnou zapisuje primární výstupy do vektoru z instance třídy Fsim. Zrušen byl také ukazatel na pointer begnet, který

ukazoval na první ze spojového seznamu hradel spojených pomocí ukazatele next ve struktuře GATE, a který sloužil jako dočasné uložiště pro hradla, než se spočítají a uloží do staticky alokovaného pole net. Toto v nové verzi není nutné, protože net je tady dynamicky alokovaný vektor a data se tak do něj mohou načítat rovnou.

K načítání ze souboru se také už nepoužívá standardní struktura FILE z jazyka C, ale používá se datový proud fstream ze standardních knihoven C++. V původním programu také byly limity pro počet hradel a počty primárních vstupů a výstupů. Pokud se u velmi velkého obvodu tyto limity překročily, vypsala se uživateli hláška, že si má dané limity upravit v kódu programu a pak si ho znovu zkompilovat. Toto by pravděpodobně znemožnilo použití programu kýmkoli, kdo se v kódu programu nevyzná. Při nastavování výstupů hradel byla obyčejná smyčka for nahrazena smyčkou foreach ze standardu C++11. Protože je v nové verzi programu uložen seznam hradel ve vektoru místo v poli, může být ve funkci levelize na obvodu nahrazena vlastní funkce setřídění seznamu podle gid funkcí std::sort ze standardních knihoven C++, která by měla dávat podstatně lepší výkon než funkce původní. Jelikož jsou ve vektoru uloženy pouze ukazatele na hradla, bylo nutné pro funkci std::sort nadefinovat jednoduchou vlastní porovnávací funkci.

3.6 Změny v knihovně io

Tato knihovna slouží hlavně k nastavení oblastí bez rozvětvení a kořenů rozvětvení, v původním programu také sloužila ke statické alokaci seznamů používaných v simulaci. Po dokončení funkcí z knihovny read_cct program pokračuje na funkce z knihovny io, a to set_cct_parameters a allocate_dynamic_buffers. První z funkcí nejdříve spočítá úrovně hloubky obvodu a pak alokuje paměť pro seznamy hradel v jednotlivých hloubkách a pak alokuje paměť pro další seznamy hradel, které se používají při simulaci. Funkce allocate_dynamic_buffers pak zkontroluje, jestli jsou seznamy alokovány a pokud ne pokusí se je alokovat. V nové verzi je první funkce zmenšena pouze na spočítání úrovní hloubky a inicializování jednotlivých vektorů pro dané hloubky a druhá funkce není vůbec použita, protože všechny seznamy jsou alokovány dynamicky. Funkce set_dominator a set_unique_path zůstaly kromě přepracování pro použití kontejnerů téměř nezměněny, až na makro Dschedule_output které bylo přepracováno na funkci stejného jména, nemohlo dojít k problémům s

typovou bezpečností a kvůli lepšímu výkonu jsou zde použité smyčky foreach ze standardu C++11.

3.7 Změny v knihovně define_fault_list

Tato knihovna v programu zajišťuje vytvoření seznamu poruch, které se budou následně testovány, a to buď načtením testovaných poruch ze souboru, nebo vytvořením kompletního seznamu poruch, kde je z každé třídy ekvivalence vybrána pouze jedna porucha. Ve funkci readfaults pro načítání poruch ze souboru jsou pouze menší změny, týkající se především změn typů proměnných na dynamická pole a několik maker pro určování typu načítaných znaků bylo změněno na funkce z důvodu typové bezpečnosti. Je zde také využito několik funkcí třídy string, jako například jeho funkce substring, nebo možnost dynamicky přidávat znaky na konec řetězce. Nastavení ukazatelů na předchozí a následující poruch bylo odstraněno, protože bylo nahrazeno dynamickým polem poruch na hradle.

Funkce pro vytváření seznamu poruch je také změněna jen málo, přesto mi při přepracování dělala problémy, hlavně kvůli způsobu, jakým je načtena první porucha, která se tam víceméně vytváří jen jako zástupná hodnota pro nastavení následující poruchy a nemá v sobě žádné informace a na konci smyčky je smazána. Trvalo mi dlouho, než jsem pochopil jakým způsobem je právě s touto zástupnou hodnotou zacházeno a jak ji správně smazat. Z této funkce jsou pak odstraněna nastavení ukazatelů na příští a předchozí poruchu a je přepracována pro nová dynamická pole.

3.8 Změny v knihovně pio

Tato knihovna slouží k načítání testovacích vektorů ze souboru a to buďto nestreamovaného vstupu, což se už moc nepoužívá z důvodu velikosti načítaných souborů, anebo z komprimovaného streamovaného vstupu. Tato knihovna v původním programu používala statické proměnné k uchování dat mezi jednotlivými běhy funkce pro načítání vektorů. V nové verzi programu je tato knihovna převedena na třídu a statické proměnné jsou změněny na vnitřní proměnné této funkce. Tato třída má celočíselnou proměnnou, určující počet již zpracovaných vektorů. Také je zde použita jednoduchá implementace kruhového bufferu, která je jak v původní, tak i v nové verzi implementována pomocí staticky alokovaného pole, protože tento buffer má naprosto úmyslně pevnou délku. Poslední proměnnou je seznam testovacích dat, ten je v původní

verzi implementován jako staticky alokované pole znaků a v převedené verzi je pro tento seznam použit kontejner vektor.

V této knihovně je také několik funkcí, týkajících se výpisu výsledků do výstupního souboru. Ty ale nejsou v ani v původním programu vůbec využity a o výpis výsledků se stará knihovna print.

3.9 Změny v knihovně ppsfp

Tato knihovna se zabývá simulací samotnou. První použitá funkce z této knihovny je funkce pfault_free_simulation. Ta se při každé smyčce simulace provádí, aby se nastavily na hradlech bezporuchové hodnoty pro daný set testovacích vektorů. V původním programu tato funkce pro vyhodnocování hodnot jednotlivých hradel používala složitá, někdy i více než deset řádků dlouhá makra. Tato makra byla v nové verzi přepracována do vlastních funkcí, které zajišťují typovou bezpečnost a zlepšují přehlednost kódu.

Druhá použitá funkce je funkce Fault1_Simulation, která zajišťuje simulaci jednotlivých poruch a jejich detekování. Tato funkce zjistí pozorovatelnost jednotlivých hradel z kořene oblasti bez rozvětvení, a na primárních výstupech označí všechny pozorovatelné poruchy jako detekované. Tato funkce byla v nové verzi téměř celá zachována, jen byla přizpůsobena pro použití dynamických datových struktur, a makra, původně používaná pro počítání hodnot daných hradel, byla nahrazena funkcemi kvůli lepší typové bezpečnosti.

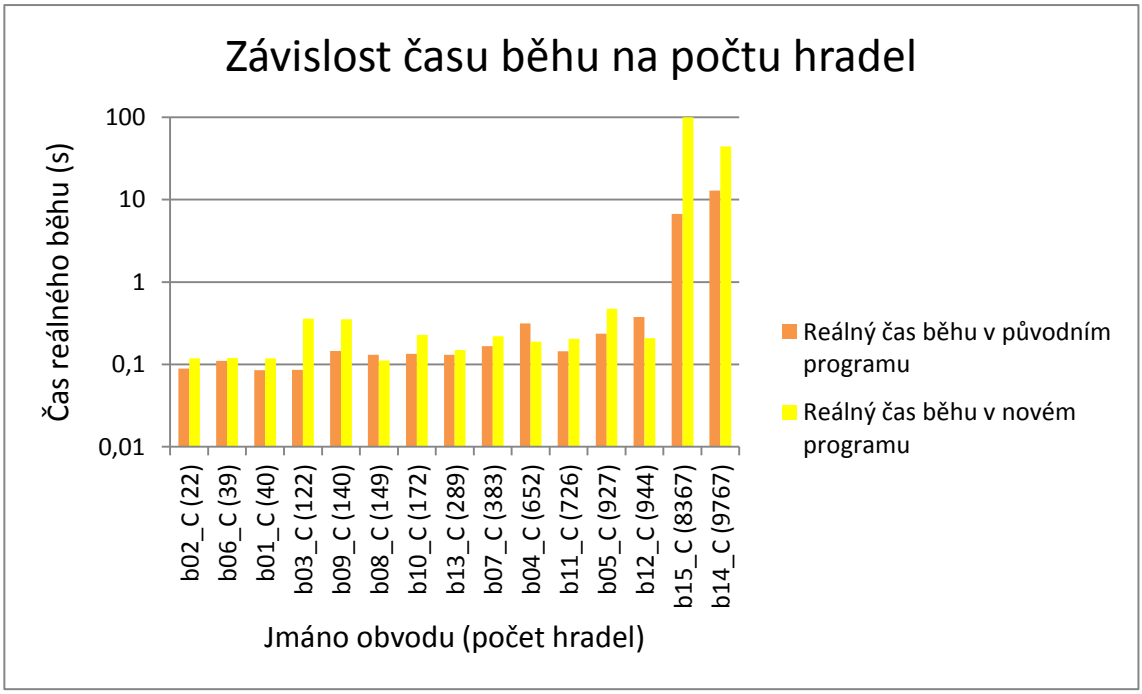
4 Testování

Podle zadání bylo výsledný program nutné otestovat na správnost jeho výsledků. K tomuto účelu mi byla přidělena sada obvodů, skládající se z souborů netlistů .bench, souborů se streamovaným testovacím vstupem .teststream a souborů .faults s kompletními sadami poruch pro dané obvody. Program jsem pak nejprve spouštěl s nastavením pro vytváření vlastního seznamu poruch, a poté s načítáním poruch ze souboru.

Z tabulky srovnání výsledků obou verzí je patrné, že se výsledky obou verzí programů liší pouze minimálně a to počtem desetinných míst, na které vypisuje funkce pro výpis. Výpis do souboru v původní verzi zajišťuje funkce fprintf, v nové verzi je použit operátor<< třídy fstream. Ostatní hodnoty, které nejsou v tabulce zobrazeny, jako

například absolutní počet detekovaných poruch, počty primárních vstupů a výstupů a počet aplikovaných vektorů, se u nové verze programu také shodují s výsledky původního programu. Z tabulky je také zřejmé, že zatímco u malých obvodů je rozdíl v délce běhu programu malý, se zvyšujícím se počtem hradel se také zvyšuje rozdíl v časech mezi těmito dvěma verzemi.

Název obvodu	Počet hradel v původním programu	Počet hradel v novém programu	Celkový Počet poruch v původním programu	Celkový Počet poruch v novém programu	Procento pokrytí v původním programu	Procento pokrytí v novém programu	Reálný čas běhu v původním programu	Reálný čas běhu v novém programu
b01_C	40	40	118	118	100.000%	100%	0,085s	0,119s
b02_C	22	22	64	64	100.000%	100%	0,089s	0,120s
b03_C	122	122	394	394	100.000%	100%	0,086s	0,119s
b04_C	652	652	1684	1684	98.872%	98.8717%	0,314s	0,359s
b05_C	927	927	2444	2444	77,823%	77,8232%	0,236s	0,354s
b06_C	39	39	136	136	100.000%	100%	0,110 s	0,111 s
b07_C	383	383	1090	1090	99.450%	99.4495%	0,166 s	0,228 s
b08_C	149	149	452	452	100.000%	100%	0,131 s	0,149 s
b09_C	140	140	405	405	100.000%	100%	0,145 s	0,220 s
b10_C	172	172	517	517	100.000%	100%	0,134 s	0,189 s
b11_C	726	726	1740	1740	96.264%	96.2644%	0,144 s	0,204 s
b12_C	944	944	2872	2872	100.000%	100%	0,377 s	0,475 s
b13_C	289	289	852	852	96.948%	96.9484%	0,131 s	0,209 s
b14_C	9767	9767	22802	22802	99.281%	99.2808%	12,9 s	99,28 s
b15_C	8367	8367	21988	21988	96.353%	96.3526%	6,7 s	44,07 s
b17_C	30777	30777	76625	76625	97.389%	97.3886%	1 min 12 s	45 min 30 s



Obrázek 8 Graf závislosti času běhu programu na počtu hradel

Z tohoto grafu je také dobře vidět, že u malých obvodů program běží téměř tak rychle jako program původní, pro složitější obvody bohužel jeho výpočetní náročnost stoupá rychleji než tomu tak je u původního obvodu.

Nový program byl také testován i s nastavením pro načítání poruch ze souboru, i při tomto nastavení program dával vždy stejné výsledky jako původní program. Testováno bylo i nastavení pro vytváření náhodných vektorů, a i při tomto nastavení program probíhal bez problémů.

5 Závěr

Účelem této práce bylo přepracovat program pro testování číslicových obvodů Fsim z programovacího jazyka C do programovacího jazyka C++. Důvodem pro tuto konverzi bylo zlepšení přehlednosti tohoto programu a snazší seznámení se s ním pro nového programátora. Překážkou tomuto je využívání množství uživatelských struktur, u kterých je minimální nebo žádný popis o jejich použití. Také se v programu vyskytuje celá řada maker, která znamenají problém s typovou kontrolou.

Dalším cílem bylo také připravit program pro další práci na něm, která by se měla týkat využití více vláken pro použití programu na vícejádrových procesorech. Program by pak mohl na více jádrech simulovat několik obvodů najednou. Hlavní překážkou v tomto cíli je, že většina proměnných nutných pro běh simulace, jsou proměnné statické a globální. Pokud by v takovém programu běželo více různých simulací najednou, simulace by si navzájem přepisovaly data a program by nemohl běžet.

Analýzou různých struktur v programu jsem došel k závěru, že se tyto struktury dělí na dva různé druhy, různé struktury, které jsou využívány k ukládání polí a struktury, které jsou specifické pro program Fsim a reprezentují součásti obvodu. První druh struktur často může být nahrazen kontejnery ze standardních knihoven jazyka C++, které zjednoduší a zpřehlední celý kód, například tím, že se jedná o standardní součásti C++ a tudíž jsou jejich použití a možnosti jasné a velice dobře popsané. U těchto tříd se také není nutné se starat o jim přidělenou paměť, kterou si tyto objekty spravují samy. Tyto struktury byly nejčastěji nahrazeny kontejnerem vektor, který spojuje rychlost procházení pole, rychlý přístup ke kterémukoli záznamu a dynamické zvětšování a zmenšování. Použit byl také kontejner list, který v programu dovoluje proměnnou používat jednoduše jako zásobník, ale i jako frontu. Posledním použitým kontejnerem by kontejner map, kterým je nahrazena jedna celá knihovna původního programu.

Druhý druh struktury byly struktury zastupující hradla a poruchy v obvodu. Tyto struktury byly přepracovány na samostatné třídy a zapouzdřeny, kvůli řízení přístupu k proměnným a zlepšení přehlednosti kódu.

Problém globálních a statických proměnných byl vyřešen vytvořením třídy Fsim, která zapouzdřuje celou simulaci a všechny její proměnné. Instance této třídy se pak

vytvoří ve funkci main a obstará celou simulaci. Při vícevláknovém běhu by si tak jednotlivé instance neměly mít možnost navzájem si přepisovat své vnitřní proměnné.

Nakonec bylo nutné program otestovat na správnost výsledků vůči původní verzi. Program byl testován s různými nastaveními a pokaždé dával stejný výsledek jako původní verze programu. Pro menší obvody měl nový program podobný čas běhu jako původní verze, ale pro větší obvody se rozdíl mezi původní verzí a novou verzí zvětšoval.

Výsledkem této práce tedy je plně funkční program Fsim přepsaný do jazyka C++, který využívá standardní knihovny jazyka C++, které zpřehledňují kód. Odstraněna je také velká většina maker z původního programu. Všechny proměnné řídící simulaci jsou zapouzdřeny ve třídě Fsim, a tak by měl program být připraven na přepracování do vícevláknového programu. Pokud se přepracovaný program překládá v příkazové řádce, je nutné použít příkaz g++ namísto gcc, jinak gcc hlásí chybu linkeru. Také je nutné při překladu použít standard C++11, jelikož program využívá některé jeho aspekty.

Seznam použité literatury

- Hlavička, J.: Diagnostika a spolehlivost. Praha, Vydavatelství ČVUT, 1998, ISBN 8001018466 Prata, S.: Mistrovství v C++, Computer press, 2007, EAN 9788025117491
- TEHRANIPOOR, Mohammad. ELECTRICAL AND COMPUTER ENGINEERING UNIVERSITY OF CONNECTICUT. Fault Simulation. [cit. 2014-09-03]. Dostupné z: http://www.engr.uconn.edu/~tehrani/teaching/test/05_Fault%20Simulation.pdf
- AMIRI, Moslem a Vaclav PRENOSIL. EMBEDDED SYSTEMS LABORATORY FACULTY OF INFORMATICS, Masaryk University Brno, Czech Republic. Digital Systems Testing: Fault Simulation Applications and Methods. 2013. [cit. 2014-09-01]. Dostupné z: http://www.fi.muni.cz/~xamiri/PA175/PA175_09.pdf
- NEUMANN, Petr. UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ – FAKULTA APLIKOVANÉ INFORMATIKY. DIAGNOSTIKA V ELEKTRONICE: AUTOMATIZACE GENEROVÁNÍ TESTŮ. 2013. [cit. 2014-08-20]. Dostupné z: http://www.utb.cz/file/41808_1_1/
- BISWAS, Santosh a Jasindra Kr. DEKA. DEPT. OF COMPUTER SCIENCE & ENGG. INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI GUWAHATI - 781039, Assam, India. VLSI Design Verification and Test[online]. [cit. 2014-08-08]. Dostupné z: http://www.nptel.ac.in/courses/106103016/24
- SCHEFFER, Lou, Luciano LAVAGNO a Grant MARTIN. EDA for IC implementation, circuit design, and process technology. Boca Raton, FL: CRC Taylor, 2006, 1 v. (various pagings). ISBN 978-084-9379-246.
- Prata, S.: Mistrovství v C++, Computer press, 2007, EAN 9788025117491

Seznam obrázků